Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2020:4
ISBN 100-

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

[1]

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

[1]

LIST OF PAPERS

1. ***WebAssembly Diversification for Malware Evasion***
Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023
`https://www.sciencedirect.com/science/article/pii/S0167404823002067`

2. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry

3. ***Multi-Variant Execution at the Edge***
Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

4. ***CROW: Code Diversification for WebAssembly***
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), MADWeb
`https://doi.org/10.14722/madweb.2021.23004`

5. ***Superoptimization of WebAssembly Bytecode***
Javier Cabrera-Arteaga,Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs
`https://doi.org/10.1145/3397537.3397567`

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
Javier Cabrera-Arteaga,Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)
`https://doi.org/10.1145/3358504.3361228`

ACKNOWLEDGEMENT

ACRONYMS

List of commonly used acronyms:

Wasm	WebAssembly

Contents

Part I

Thesis

01 INTRODUCTION

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.1 Background

TODO Motivate with the open challenges.

1.2 Problem statement

TODO Problem statement TODO Set the requirements as R1, R2, then map each contribution to them.

1.3 Automatic Software diversification requirements

1. **1:** TODO Requirement 1

1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

[0] Comp. time 2023/10/03 14:18:41

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1: *Mapping of the contributions to the research papers appended to this thesis.*

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

1.5 Summary of research papers

P1: Superoptimization of WebAssembly Bytecode.

P2: CROW: Code randomization for WebAssembly bytecode.

P3: Multivariant execution at the Edge.

P4: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

P5: WebAssembly Diversification for Malware evasion.

1.6 Thesis outline

02 BACKGROUND AND STATE OF THE ART

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the four scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?**] in 2017. Wasm is designed to be fast, portable, self-contained and secure, and it promises to outperform JavaScript execution. Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version[2].

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [**?** **?**]. Consequently, in 2019, the Bytecodealliance proposed WebAssembly System Interface (WASI) [**?**]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [**?**], making it suitable standalone and backend execution scenarios [**?** **?**].

2.1.2 Generating WebAssembly programs

WebAssembly programs are pre-compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For

[0] Comp. time 2023/10/03 14:18:41

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

In Listing 2.1 and Listing 2.2, we illustrate a C program and its corresponding Wasm binary. The C function includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses. The Wasm code in Listing 2.2 displays the textual format of the generated Wasm (Wat)[3].

```c
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.1: *Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.*

[3] The WAT text format is mostly for human readability and for low-level manual modification.

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
  (type (;0;) (func (param f32) (result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
  (import "env" "ftoi" (func $ftoi (type 0)))
; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
  (@custom "name" "...")
; Code section: 0x03 0x00 0x00 0x00 0x5B... ;
  (func $main (type 2) (result i32)
    (local i32 i32)
    i32.const -1000
    local.set 0
    block  ;label = @1;
      loop  ;label = @2;
        i32.const 0
        local.get 0
        i32.add
        i32.load
        local.tee 1
        i32.const 101
        i32.ge_s
        br_if 1 ;@1;
        local.get 0
        i32.const 4
        i32.add
        local.tee 0
        br_if 0 ;@2;
      end
      i32.const 0
      return
    end
    f32.const 0x1.9147aep+3
    call $ftoi
    local.get 1
    i32.add)
; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
  (memory (;0;) 1)
; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
  (global (;4;) i32 (i32.const 1000))
; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
  (export "memory" (memory 0))
  (export "A" (global 2))
; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
  (data $data (0) "\00\00\00\00...")
; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
  (@custom "producers" "...")
)
```

Listing 2.2: *Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.*

■ 2.1.3 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A Wasm binary contains sections of 13 types, each with a specific semantic role and placement within the module. Each section is optional, where an omitted section is considered empty. In the following text, we summarize each one of the 13 types of Wasm sections, providing their name, ID, and purpose. In addition, some sections are annotated as comments in the Wasm code in Listing 2.2.

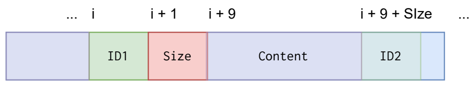

Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

Custom Section (`00`) : Comprises two parts: the section name and arbitrary content. Primarily used for storing metadata, such as the compiler used to generate the binary (see lines 9 and 48 of Listing 2.2). This type of section has no order constraints with other sections and is optional. Compilers usually skip this section when consuming a WebAssembly binary.

Type Section (`01`) : Contains the function signatures for functions declared or defined within the binary (see lines 3 to 6 in Listing 2.2). It must occur only once in a binary. It can be empty.

Import Section (`02`) : Lists elements imported from the host, including functions, memories, globals, and tables (see line 8 in Listing 2.2). It must occur only once in a binary. It can be empty.

Function Section (`03`) : Details functions defined within the binary. It essentially maps Type section entries to Code section entries. The text format already maps the function index to its name, as shown in lines 12 to 38 of Listing 2.2. This section must occur only once in a binary and, it can be empty.

Table Section (`04`) : Groups functions with identical signatures to control indirect calls. It must occur only once in a binary. It can be empty. The example code in Listing 2.2 does not include a Table Section.

Memory Section (`05`) : Specifies the number and initial size of unmanaged linear memories (see line 40 in Listing 2.2). It must occur only once in a binary. It can be empty.

Global Section (`06`) : Defines global variables as managed memory for use and sharing between functions in the WebAssembly binary (see line 42 of Listing 2.2). It must occur only once in a binary. It can be empty.

Export Section (`07`) : Declares elements like functions, globals, memories, and tables for host engine access (see lines 44 and 45 of Listing 2.2). It must occur only once in a binary. It can be empty.

Start Section (`08`) : Designates a function to be called upon binary readiness, initializing the WebAssembly program state before executing any exported functions. It must occur only once in a binary. It can be empty. The example code in Listing 2.2 does not include a Start Section, i.e. there is no function to call when the binary is initialized.

Element Section (`09`) : Contains elements to initialize the binary tables. It must occur only once in a binary. It can be empty. The example code in Listing 2.2 does not include an Element Section.

Code Section (`10`) : Contains the body of functions defined in the Function section. Each entry consists of local variables used and a list of instructions (see lines 12 to 38 in Listing 2.2). It must occur only once in a binary. It can be empty.

Data Section (`11`) : Holds data for initializing unmanaged linear memory. Each entry specifies the offset and data to be placed in memory (see line 47 in Listing 2.2). It must occur only once in a binary. It can be empty.

Data Count Section (`12`) : Primarily used for validating the Data Section. If the segment count in the Data Section mismatches the Data Count, the binary is considered malformed. The example code in Listing 2.2 does not include a Data Count Section. It must occur only once in a binary. It can be empty.

Due to its organization into a contiguous array of sections, a Wasm binary can be processed efficiently. For example, this structure allows compilers to speed up the compilation process through parallel parsing or just by ignoring *Custom Sections*. Additionally, the use of the LEB128[4] encoding of instructions of the *Code Section* further compacts the binary. As a result, Wasm binaries are not only fast to process but also quick to transmit over a network.

■ 2.1.4 WebAssembly's runtime structure

The WebAssembly runtime structure is described in the WebAssembly specification by enunciating 10 key components: the Store, Module Instances, Table Instances, Export Instances, Import Instances, the Execution Stack, Memory Instances, Global Instances, Function Instances and Locals. These components are particularly significant in maintaining the state of a WebAssembly program during its execution. In the following text, we provide a

[4] `https://en.wikipedia.org/wiki/LEB128`

brief description of each runtime component. Notice that, the runtime structure is an abstraction that serves to validate the execution of a Wasm binary.

Store : The WebAssembly store represents the global state and is a collection of instances of functions, tables, memories, and globals. Each of these instances is uniquely identified by an address, which is usually represented as an i32 integer.

Module Instances : A module instance is a runtime representation of a loaded and initialized WebAssembly module. It contains the runtime representation of all the definitions within a module, including functions, tables, memories, and globals, as well as the module's exports and imports.

Table instances : A table instance is a vector of function elements. WebAssembly tables are used to support indirect function calls. For example, it allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is in charge of populating the static table of functions.

Export Instances : Export instances represent the functions, tables, elements, globals or memories that are exported by a Wasm binary to the host environment.

Import Instances : Import instances represent the functions, tables, elements, globals or memories that are imported into a module from the host environment.

The Execution Stack holds typed values and control frames, with control frames handling block instructions, loops, and function calls. Values inside the stack can be of the only static types allowed in Wasm 1.0, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. Therefore, abstract types, such as classes, objects, and arrays, are not natively supported. Instead, during compilation, such types are transformed into primitive types and stored in the linear memory.

Memory Instances represent the unmanaged linear memory of a WebAssembly program, consisting of a contiguous array of bytes. Memory instances are accessed with `i32` pointers (integer of 32 bits). Memory instances are usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiates the WebAssembly module and the binary itself.

Global Instances : A global instance is a global variable with a value and a mutability flag, indicating whether the global can be modified or is immutable. Global variables are part of the managed data, i.e., their allocation and memory placement are managed by the host engine. Global variables are only accessible by their declaration index, and it is not possible to dynamically address them.

Locals : Locals are mutable variables that are local to a specific function instance, i.e. locals are only accessible through their index related to the executing function instance. As globals, locals are part of the managed data.

Definition 1. *Along with this dissertation, as the work of Lehmann et al. [?], we refer to managed and unmanaged data to differentiate between the data that is managed by the host engine and the data that is managed by the WebAssembly program respectively.*

Function Instances : A function instance groups locals and a function body. Locals are typed variables that are local to a specific function invocation as previously discussed. The function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the stack, writes to the stack, or modifies the control flow of the function. Recalling the example Wasm binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 7 and Line 32 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add`. As the listing also shows, instructions are annotated with a numeric type.

■ 2.1.5 WebAssembly's control flow

In WebAssembly, a defined function instructions are organized into blocks, with the function's starting point serving as the root block. Unlike traditional assembly code, control flow structures in Wasm jump between block boundaries rather than arbitrary positions within the code. Each block might specify the required stack state before execution and the resulting stack state after its instructions have run. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before executing the block's instructions. Blocks in Wasm are explicit, indicating, where they start and end. By design, each block cannot reference or execute code from outer blocks.

Control flow within a function is managed through three types of break instructions: unconditional break, conditional break, and table break. Importantly, each break instruction is limited to jumping to one of its enclosing blocks. Unlike standard blocks, where breaks jump to the end of the block, breaks within a loop block jump to the block's beginning, effectively restarting the loop. To illustrate this, Listing 2.3 provides an example comparing a standard block and a loop block in a Wasm function.

```
block
   block
      br 1    ; Jump instructions
             are annotated with the
             depth of the block they
              jump to;
   end
end
...
```

```
loop
   ...
   br 0    ;first-order break;
   ...
end     ; end instructions break
        the block and jump to next
        instruction;
...
```

Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. For the purposes of this dissertation, we introduce a specific term to describe a particular kind of break within loops:

Definition 2. *Break instructions within loops that effectively jump to the loop's beginning are termed* first-order breaks*.*

■ 2.1.6 WebAssembly's ecosystem

TODO Split in two sections. Do a new section on WebAssembly analysis. **TODO** Other WebAssembly tools section.

WebAssembly programs are designed for execution in host environments such as web browsers. Though the execution of a WebAssembly program might be considered its final lifecycle stage, the WebAssembly ecosystem is far from simplistic. It comprises multiple stakeholders and a rich array of tools that cater to various needs [**?**]. In Figure 2.2 we simplify the ecosystem landscape by separating it into producers, consumers and major stakeholders categories. In the subsequent text, we describe the WebAssembly ecosystem.

Producers , such as compilers, transform source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release[5], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[6]. In parallel developments, the KMM framework[7] has incorporated WebAssembly as a compilation target, and the Javy approach[8] focuses on encapsulating JavaScript code within isolated WebAssembly binaries. This latter is achieved by porting both the engine and the source code into a

[5]`https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`
[6]A subset of the TypeScript language
[7]`https://kotlinlang.org/docs/wasm-overview.html`
[8]`https://github.com/bytecodealliance/javy`

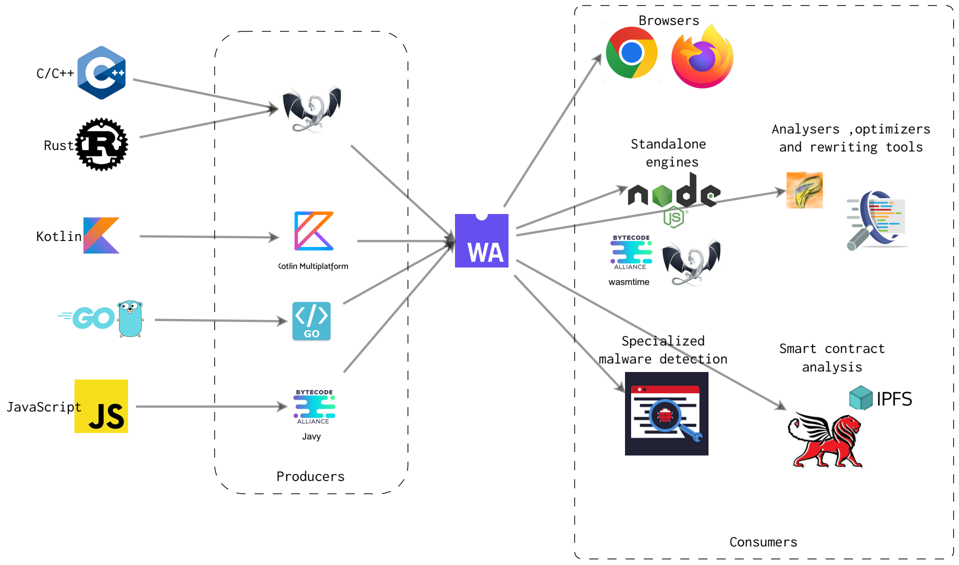

Figure 2.2: *WebAssembly's ecosystem landscape separated into producers and consumers. For the sake of simplicity we do not include each tool mentioned in this chapter.*

secure WebAssembly environment. Similarly, Blazor also enables the compilation of C code into WebAssembly binaries for browser execution[9].

From a security standpoint, WebAssembly programs are designed without a standard library and are prohibited from direct interactions with the operating system. Instead, the host environment offers a predefined set of functions that can be imported into the WebAssembly program. It falls upon the producers to specify which functions from the host environment will be imported by the WebAssembly application.

Consumers encompass tools that undertake the tasks of validating, analyzing, optimizing, transpiling to machine code, and executing WebAssembly binaries, e.g. browser clients. In the text that follows, we dissect them into specific categories and their respective domains of application. Notice that, while some tools are designed for a specific domain, others are more general-purpose and might encapsulate more than one task in the WebAssembly ecosystem. For example, this is the case of browsers and standalone engines, which in one way or the other perform each one of the previous tasks.

Browser engines like V8[10] and SpiderMonkey[11] are at the forefront of executing

[9]`https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor`
[10]`https://chromium.googlesource.com/v8/v8.git`
[11]`https://spidermonkey.dev/`

WebAssembly binaries in browser clients. These engines leverage Just-In-Time (JIT) compilers to convert WebAssembly into machine code. This translation is typically a straightforward one-to-one mapping, given that WebAssembly is already an optimized format closely aligned with machine code, as previously discussed in Subsection 2.1.3. For example, V8 just employs quick, rudimentary optimizations, such as constant folding and dead code removal, to guarantee fast readiness for a Wasm binary to execute.[12]

Standalone engines: WebAssembly has expanded beyond browser environments, largely due to the WASI[**?**]. It standardizes the interactions between host environments and WebAssembly modules through a POSIX-like interface. A range of standalone engines like WASM3[13], Wasmer[14], Wasmtime[15], WAVM[16], and Sledge[**?**] have emerged to support WebAssembly and WASIc. In a similar vein, Singh et al.[**?**] introduced a virtual machine for WebAssembly tailored for Arduino-based devices. Salim et al.[**?**] proposed TruffleWasm, an implementation of WebAssembly hosted on Truffle and GraalVM. Additionally, SWAM[17] stands out as WebAssembly interpreter implemented in Scala. Finally, WaVe[**?**] offers a WebAssembly interpreter featuring mechanized verification of the WebAssembly-WASI interaction with the underlying operating system.

■ 2.1.7 WebAssembly binary analysis

Static and dynamic analysis, optimization and validation: As the WebAssembly ecosystem continues to grow, the need for robust tools to ensure its security and reliability has increased. To address this, a variety of tools have been developed that employ different strategies to identify vulnerabilities in WebAssembly programs. Tools like Wassail[**?**], SecWasm[**?**], Wasmati[**?**], WasmA[**?**], and Wasp[**?**] leverage techniques such as information flow control, code property graphs, control flow analysis, and concolic execution. Similarly, VeriWasm[**?**] stands out as a static offline verifier specifically designed for native x86-64 binaries compiled from WebAssembly. In the realm of dynamic analysis, tools like TaintAssembly[**?**], Wasabi[**?**], and Fuzzm[**?**] offer similar functionalities. Hybrid methods have also gained traction, with tools like CT-Wasm[**?**] enabling the verifiably secure implementation of cryptographic

[12]This analysis was corroborated through discussions with the V8 development team and through empirical studies in one of our contributions[?]

[13]`https://github.com/wasm3/wasm3`

[14]`https://wasmer.io/`

[15]`https://github.com/bytecodealliance/wasmtime`

[16]`https://github.com/WAVM/WAVM`

[17]`https://github.com/satabin/swam`

algorithms in WebAssembly. Binaryen[18] serves as a comprehensive toolkit for WebAssembly binary manipulation, including validation, optimization, and compilation to machine code. Stiévenart and colleagues have introduced a dynamic approach to slice WebAssembly programs based on Observational-Based Slicing (ORBS)[? ?]. Finally, Wafl[?] extends AFL++ to perform coverage-based fuzzing on WebAssembly binaries.

Specialized Malware Detection In niche areas like cryptomalware detection, tools like MineSweeper[?], MinerRay[?], and MINOS[?] utilize static analysis through machine learning techniques to detect browser cryptomalwares. Conversely, tools like SEISMIC[?], RAPID[?], and OUTGuard[?] seek the same goal with dynamic analysis techniques. **TODO** Add VirusTotal here **TODO** Compare

Smart Contract Analysis In the field of smart contracts, static analysis tools like WANA[?], and EOSAFE[?] are employed to unearth vulnerabilities in WebAssembly smart contracts. Dynamic analysis tools in this sphere include EOSFuzzer[?] and wasai[?]. Similarly, Manticore[19] supports the symbolic execution of Wasm smart contracts.

Binary rewriting tools and obfuscators The landscape for tools that can modify, obfuscate, or enhance WebAssembly binaries for various has increased. For instance, BREWasm[?] provides a comprehensive static binary rewriting framework specifically designed for WebAssembly. Wobfuscator[?] takes a different approach, serving as an opportunistic obfuscator for Wasm-JS browser applications. Madvex[?] focuses on modifying WebAssembly binaries to evade malware detection, with its approach being limited to alterations in the code section of a WebAssembly binary. Additionally, WASMixer[?] obfuscates WebAssembly binaries, by including memory access encryption, control flow flattening, and the insertion of opaque predicates.

2.1.8 WebAssembly opportunities

TODO Emphasize what is missing. DO not talk about monoculture, neither diversification. Talks about runtime properties, side-channels, etc. **TODO** Put things into context.

As outlined, WebAssembly is deterministic, well-typed, and follows a structured control flow, making it easier for compilers and engines to sandbox its execution[?]. Despite its robust design, the WebAssembly's ecosystem is still nascent and faces security vulnerabilities, both for WebAssembly consumers and with the WebAssembly binaries themselves. For instance, Genkin et al.

[18]`https://github.com/WebAssembly/binaryen`
[19]`https://github.com/trailofbits/manticore/tree/master/manticore`

demonstrated that WebAssembly could be exploited to exfiltrate data using cache timing-side channels[**?**]. Lehmann et al. and Stiévenart and colleagues show that vulnerabilities in C/C++ source code could propagate to WebAssembly binaries[**?** **?**].

■ 2.2 Software diversification

■ 2.2.2 Generating Software Diversification

■ 2.2.3 Variants generation

■ 2.2.4 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. `https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13`

■ 2.2.5 Defensive Diversification

■ 2.2.6 Offensive Diversification

03 AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

The process of generating WebAssembly binaries starts with the original source code, which is then processed by a compiler to produce a WebAssembly binary. This compiler is generally divided into three main components: a frontend that converts the source code into an intermediate representation, an optimizer that refines this representation for performance, and a backend that compiles the final WebAssembly binary. For example, LLVM uses this architecture, supporting several programming languages like C, C++, and Rust for several backed architectures, including WebAssembly. On the other hand, Software Diversification, a preemptive security measure, can be integrated at various stages of this compilation process. However, applying diversification at the front-end has its limitations, as it would need a unique diversification mechanism for each language compatible with WebAssembly. Conversely, diversification at later compiler stages, such as the optimizer or backend, offers a more practical alternative.

Significantly, a study by Hilbig et al. reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. This makes the latter stages of the LLVM compiler an ideal point for introducing practical Wasm diversification techniques. Our compiler-based strategies, represented in red and green in Figure 3.1, introduce a diversifier component into the LLVM pipeline. This component generates LLVM IR variants, thereby creating artificial software diversity for WebAssembly. Specifically, we propose two tools: CROW, which generates WebAssembly program variants, and MEWE, which packages these variants to enable multivariant execution [**?**]. Alternatively, diversification can be directly applied to the WebAssembly binary, offering a language and compiler agnostic approach. Our binary-based strategy, WASM-MUTATE, represented in blue in Figure 3.1, employs rewriting rules on an e-graph data structure to generate a variety of WebAssembly program variants.

This dissertation contributes to the field of Software Diversification for WebAssembly by presenting two primary strategies: compiler-based and binary-based. Within this chapter, we introduce three technical contributions:

[0] Comp. time 2023/10/03 14:18:41

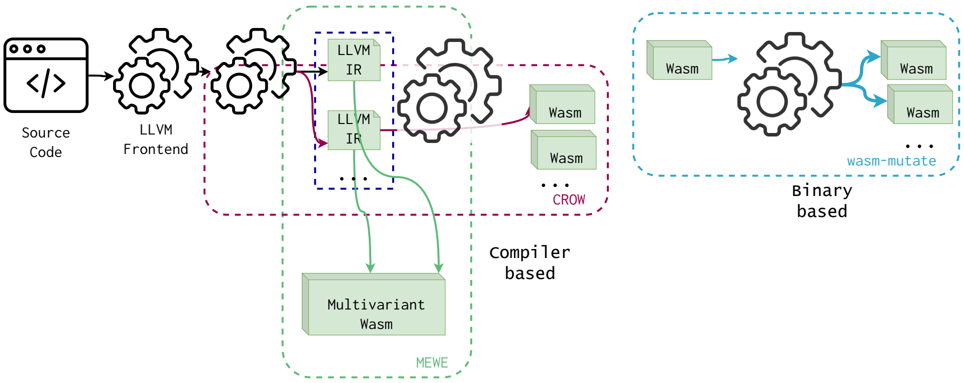

Figure 3.1: *Approach landscape containing our three technical contributions: CROW squared in red, MEWE squared in green and WASM-MUTATE squared in blue. We annotate where our contributions, compiler-based and binary-based, stand in the landscape of generating WebAssembly programs.*

CROW, MEWE, and WASM-MUTATE. We also compare these contributions, highlighting their complementary nature. Additionally, we provide the artifacts for our contributions to promote open research and reproducibility.

■ 3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?**], represented as the red squared tooling in Figure 3.1. CROW is designed to produce functionally equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combining*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combining* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

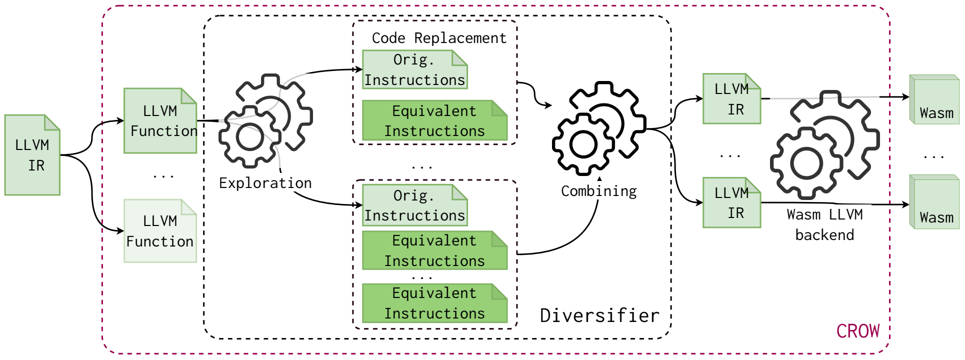

Figure 3.2: *CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them. Figure taken from [?].*

■ 3.1.2 Enumerative synthesis

The cornerstone of CROW's exploration mechanism is its code replacement generation strategy, which is inspired by the superdiversifier methodology proposed by Jacob et al. [?]. The search space for generating variants is delineated through an enumerative synthesis process, which systematically produces all possible code replacements for each instruction and its data flow graph in the original program. If a code replacement is identified to perform identically to the original program, it is reported as a functionally equivalent variant. This equivalence is confirmed using a theorem solver for rigorous verification.

Concretely, CROW is developed by modifying the enumerative synthesis implementation found in Souper [?], an LLVM-based superoptimizer. Specifically, CROW constructs a Data Flow Graph for each LLVM instruction that returns an integer. Subsequently, it generates all viable expressions derived from a selected subset of the LLVM Intermediate Representation language for each DFG. The enumerative synthesis process incrementally generates code replacements, starting with the simplest expressions (those composed of a single instruction) and gradually increasing in complexity. The exploration process continues either until a timeout occurs or the size of the generated replacements exceeds a predefined threshold.

Notice that the search space increases exponentially with the size of the language used for enumerative synthesis. To mitigate this issue, we prevent CROW from synthesizing instructions without correspondence in the Wasm backend, effectively reducing the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more program variants.

For example, Souper prevents the generation of commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

Leveraging the ascending nature of its enumerative synthesis process, CROW is capable of creating variants that may outperform the original program in both size and efficiency. For instance, the first functionally equivalent transformation identified is typically the most optimal in terms of code size. This approach offers developers a range of performance options, allowing them to balance between diversification and performance without compromising the latter. CROW applies these code transformations incrementally. For instance, if a suitable replacement is identified that can be applied at N different locations in the original program, CROW will generate N distinct program variants, each with the transformation applied at a unique location. This approach leads to a combinatorial explosion in the number of available program variants, especially as the number of possible replacements increases.

The last stage at CROW involves a custom Wasm LLVM backend, which generates the Wasm programs. For it, we remove all built-in optimizations in the LLVM backend that could reverse Wasm variants, i.e., we disable all optimizations in the Wasm backend that could reverse the CROW transformations.

■ 3.1.3 Constant inferring

CROW, inherently adds a new transformation strategy that leads to more Wasm program variants, *constant inferring.* In concrete, Souper infers pieces of code as a single constant assignment and this is ported to CROW. This strategy mostly focuses on variables that are used to control branches. After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification.

Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smallest number that fits with the Babbage condition in Line 4.

```
int babbage() {
    int current = 0,
        square;
    while ((square=current*current) % 1000000 != 269696) {
        current++;
    }
    printf ("The number is %d\n", current);
    return 0 ;
}
```

Listing 3.1: *Babbage problem. Taken from [?].*

```
1  int babbage() {
2      int current = 25264;
3
4
5
6
7
8
9      printf ("The number is %d\n", current);
10     return 0 ;
11 }
```

Listing 3.2: *Constant inferring transformation over the original Babbage problem in Listing 3.1. Taken from [?].*

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the `while`-loop because the loop count is too large. CROW deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime[1]

3.1.4 CROW instantiation

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

```
int f(int x) {
    return 2 * x + x;
}
```

Listing 3.3: *C function that calculates the quantity $2x + x$.*

[1]Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR.

```
define i32 @f(i32) {

  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

  ret i32 %3
}

define i32 @main() {
  %1 = tail call i32 @f(
        i32 10)
  ret i32 %1
}
```

```
Replacement candidates
      for code_1

%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2

%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.4: *LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.*

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

Listing 3.5: *Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.*

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping.*

```
func $f (param i32) (result i32)
   local.get 0
    i32.const 2
    i32.mul
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.shl
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    local.get 0
    i32.add
    local.get 0
    i32.add

func $f (param i32) (result i32)
    local.get 0
    i32.const 3
    i32.mul
```

Listing 3.6: *Wasm program variants generated from program Listing 3.3.*

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code.

Contribution paper and artifact

CROW fully presented in Cabrera-Arteaga et al. "CROW: Code Randomization of WebAssembly" *at proceedings of NDSS, Measurements, Attacks, and Defenses for the Web (MADWeb) 2021* `https://doi.org/10.14722/madweb.2021.23004`.

CROW source code is available at `https://github.com/ASSERT-KTH/slumps`

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [?]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE) [?]. MEWE generates application-level multivariant binaries without changing the operating system or Wasm runtime. It creates an MVE by intermixing functions for which CROW generates variants, as illustrated by the green square in Figure 3.1. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

As illustrated in Figure 3.3, MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE,

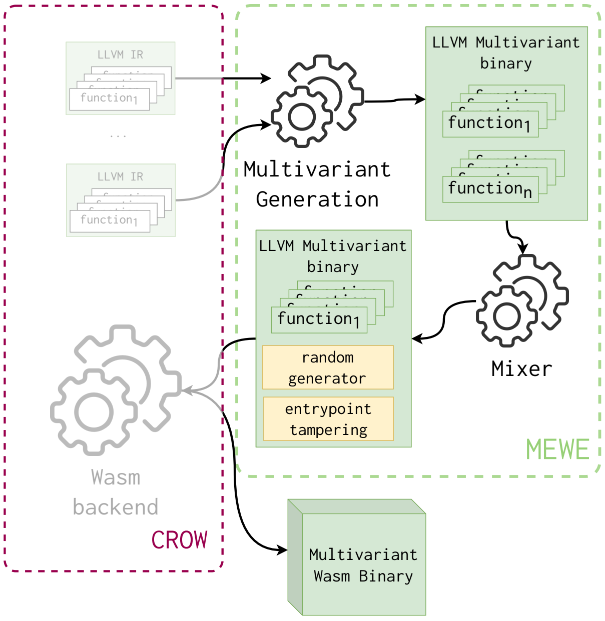

Figure 3.3: *Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed. Figure partially taken from [**?**].*

Multivariant Generation and the *Mixer*. In the *Multivariant Generation* process, MEWE gathers the LLVM IR variants created by CROW. The Mixer component, on the other hand, links the multivariant binary and creates a new entrypoint for the binary called *entrypoint tampering*. The tampering is needed in case the output of CROW are variants of the original entrypoint, e.g. the *main* function. Concretely, it wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint. The random generator is needed to perform the execution-path randomization. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. Finally, using the same custom Wasm LLVM backend as CROW, we generate a standalone multivariant Wasm binary. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

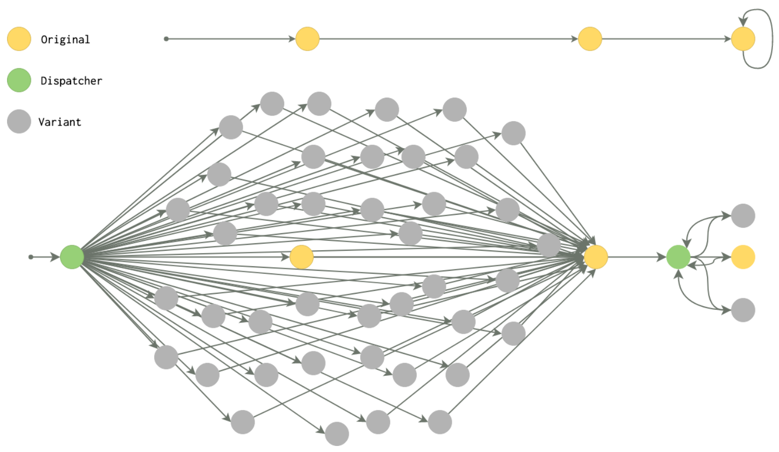

Figure 3.4: *Example of two static call graphs. At the top, is the original call graph, and at the bottom, is the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow). Figure taken from [**?**].*

3.2.2 Multivariant generation

The key component of MEWE consists of combining the variants into a single binary. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

```
 2  ; Multivariant foo wrapping ;
 3  define internal i32 @foo(i32 %0) {
 4      entry:
 5        ; It first calls the dispatcher to discriminate between the created
             variants ;
 6        %1 = call i32 @discriminate(i32 3)
 7        switch i32 %1, label %end [
 8          i32 0, label %case_43_
 9          i32 1, label %case_44_
10        ]
11      ;One case for each generated variant of foo ;
12      case_43_:
13        %2 = call i32 @foo_43_(%0)
14        ret i32 %2
15      case_44_:
16        ; MEWE can inline the body of the a function variant ;
17        %3 = <body of foo_44_ inlined>
18        ret i32 %3
19      end:
20        ; The original is also included ;
21        %4 = call i32 @foo_original(%0)
22        ret i32 %4
23  }
```

Listing 3.7: *Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4. The code is commented for the sake of understanding.*

TODO Recheck these works on the decision of using switch cases.

In Listing 3.7, we demonstrate how MEWE constructs the function dispatcher, corresponding to the rightmost green node in Figure 3.4, which handles three created variants including the original. The dispatcher function retains the same signature as the original function. Initially, the dispatcher invokes a random number generator, the output of which is used to select a specific function variant for execution (as seen on line 6 in Listing 3.7). To enhance security, we employ a switch-case structure within the dispatcher, mitigating vulnerabilities associated with speculative execution-based attacks [?] (refer to lines 12 to 19 in Listing 3.7). This approach also eliminates the need for multiple function definitions with identical signatures, thereby reducing the potential attack surface in cases where the function signature itself is vulnerable [?]. Additionally, MEWE can inline function variants directly into the dispatcher, obviating the need for redundant definitions (as illustrated on line 16 in Listing 3.7). Remarkably, we prioritize security over performance, i.e., while using indirect calls in place of a switch-case could offer constant-time performance benefits, we implement switch-case structures.

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. Notice that, the dispatcher function is constructed using the same signature as the original function. It first calls the random generator, which returns a value used to invoke a specific function variant (see line 6 in Listing 3.7). We utilize a switch-case structure in the dispatchers to prevent indirect calls, which are vulnerable to speculative execution-based attacks [?] (see lines 12 to 19 in Listing 3.7), i.e., the choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [?]. In addition, MEWE can inline function variants inside the dispatcher instead of defining them again (see line 16 in Listing 3.7). Remarkably, we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

Contribution paper and artifact

MEWE is fully presented in Cabrera-Arteaga et al. "Multi-Variant Execution at the Edge" *Proceedings of ACM, Moving Target Defense* `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

MEWE is also available as an open-source tool at `https://github.com/ASSERT-KTH/MEWE`

3.3 WASM-MUTATE: Fast and Effective Binary for WebAssembly

In this section, we introduce our third technical contribution, WASM-MUTATE [?], a tool that generates functionally equivalent variants of a WebAssembly binary input. Leveraging rewriting rules and e-graphs [?] for software diversification, WASM-MUTATE synthesizes program variants by transforming parts of the original binary. In Figure 3.1, we highlight WASM-MUTATE as the blue squared tooling.

Figure 3.5 illustrates the workflow of WASM-MUTATE, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g. an intermediate representation. Subsequently, WASM-MUTATE utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains functional equivalence to the original binary. This assurance of functional equivalence is rooted in the inherent properties of the individual rewrite rules employed.

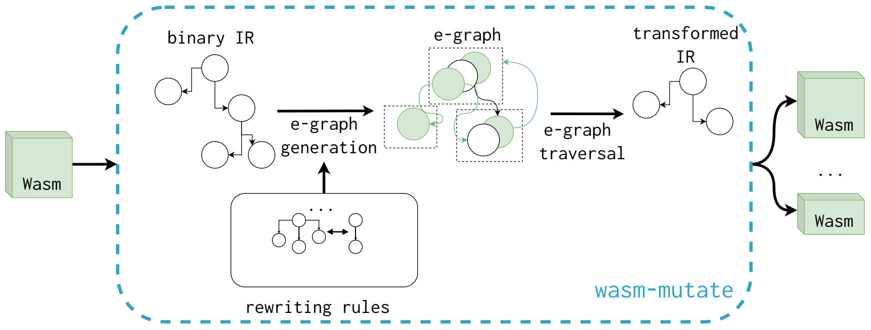

Figure 3.5: *WASM-MUTATE high-level architecture. It generates functionally equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by diversification space traversals using e-graphs.*

■ 3.3.2 WebAssembly Rewriting Rules

WASM-MUTATE contains a comprehensive set of 135 rewriting rules. Inside WASM-MUTATE, a rewriting rule is a tuple `(LHS, RHS, Cond)` where `LHS` specifies the segment of code targeted for replacement, `RHS` describes its functionally equivalent substitute, and `Cond` outlines the conditions that must be met for the replacement to take place, e.g. enhancing type constraints. WASM-MUTATE groups these rewriting rules into meta-rules depending on their target inside a Wasm binary, ranging from high-level changes affecting all binary section structure to low-level modifications within the code section. This section focuses on the biggest meta-rule implemented in WASM-MUTATE, the `Peephole` meta-rule [2].

Rewriting rules inside the *Peephole* meta-rule, operate over the data flow graph of instructions within a function body, representing the lowest level of rewriting. In WASM-MUTATE, we have implemented 125 rewriting rules specifically for this category, each one avoiding targeting instructions that might induce undefined behavior, e.g., function calls.

Moreover, we augment the internal representation of a Wasm program to bolster WASM-MUTATE's transformation capabilities through the `Peephole` meta-rule. Concretely, we augment the parsing stage in WASM-MUTATE by including custom operator instructions. These custom operator instructions are designed to use well-established code diversification techniques through rewriting rules. In practice, custom operators are only part of the rewriting rules of WASM-MUTATE. This means that, when parsing Wasm to the intermediate representation no custom operator is generated. When converting back to the WebAssembly binary format from the intermediate representation, custom

[2]For an in-depth explanation of the remaining meta-rules, refer to [**?**].

instructions are meticulously handled to retain the original functionality of the WebAssembly program. Remarkably, custom operators highlight the versatility of WASM-MUTATE as a general-purpose binary rewriting engine.

In the example below, we illustrate a rewriting rule of the `Peephole` meta-rule that leverages a custom operator to insert `nop` instructions into any WebAssembly program place, a well-known low-level diversification strategy [?], while using the `container` custom operator. When generating the Wasm variant, the `container` custom operator is removed while its operands are encoded back to Wasm in their corresponding opcodes.

```
LHS x
```

```
RHS (container (x nop))
```

3.3.3 E-Graphs traversals

We developed WASM-MUTATE leveraging e-graphs, a specific graph data structure for representing and applying rewriting rules [?]. In the context of WASM-MUTATE, a primer e-graph is constructed from the input WebAssembly program. This initial e-graph is subsequently augmented with e-nodes and e-classes derived from each one of the rewriting rules (we detail the e-graph construction process in Section 3 of [?]).

Willsey et al. highlight the potential for high flexibility in extracting code fragments from e-graphs, a process that can be recursively orchestrated through a cost function applied to e-nodes and their respective operands. This methodology ensures the functional equivalence of the derived code [?]. For instance, e-graphs solve the problem of providing the best code out of several optimization rules [?]. To extract the "optimal" code from an e-graph, one might commence the extraction at a specific e-node, subsequently selecting the AST with the minimal size from the available options within the corresponding e-class's operands. In omitting the cost function from the extraction strategy leads us to a significant property: *any path navigated through the e-graph yields a functionally equivalent code variant.*

We exploit such property to fastly generate diverse WebAssembly variants. We propose and implement an algorithm that facilitates the random traversal of an e-graph to yield functionally equivalent program variants, as detailed in Algorithm 1. This algorithm operates by taking an e-graph, an e-class node (starting with the root's e-class), and a parameter specifying the maximum extraction depth of the expression, to prevent infinite recursion. Within the algorithm, a random e-node is chosen from the e-class (as seen in lines 5 and 6), setting the stage for a recursive continuation with the offspring of the selected e-node (refer to line 8). Once the depth parameter reaches zero, the algorithm extracts the most concise expression available within the current e-class (line

3). Following this, the subexpressions are built (line 10) for each child node, culminating in the return of the complete expression (line 11).

Algorithm 1 e-graph traversal algorithm taken from [?].

```
1: procedure TRAVERSE($egraph$, $eclass$, $depth$)
2:     if depth = 0 then
3:         return smallest_tree_from(egraph, eclass)
4:     else
5:         $nodes \leftarrow egraph[eclass]$
6:         $node \leftarrow random\_choice(nodes)$
7:         $expr \leftarrow (node, operands = [])$
8:         for each $child \in node.children$ do
9:             $subexpr \leftarrow$ TRAVERSE($egraph$, $child$, $depth - 1$)
10:            $expr.operands \leftarrow expr.operands \cup \{subexpr\}$
11:        return $expr$
```

■ 3.3.4 WASM-MUTATE instantiation

Let us illustrate how WASM-MUTATE generates variant programs by using the before enunciated algorithm. Here, we use Algorithm 1 with a maximum depth of 1. In Listing 3.8 a hypothetical original Wasm binary is illustrated. In this context, a potential user has set two pivotal rewriting rules: `(x, container (x nop),)` and `(x, x i32.add 0, x instanceof i32)`. The former rule, which has been previously discussed in Subsection 3.3.2, grants the ability to append a `nop` instruction to any subexpression. Conversely, the latter rule adds zero to any numeric value .

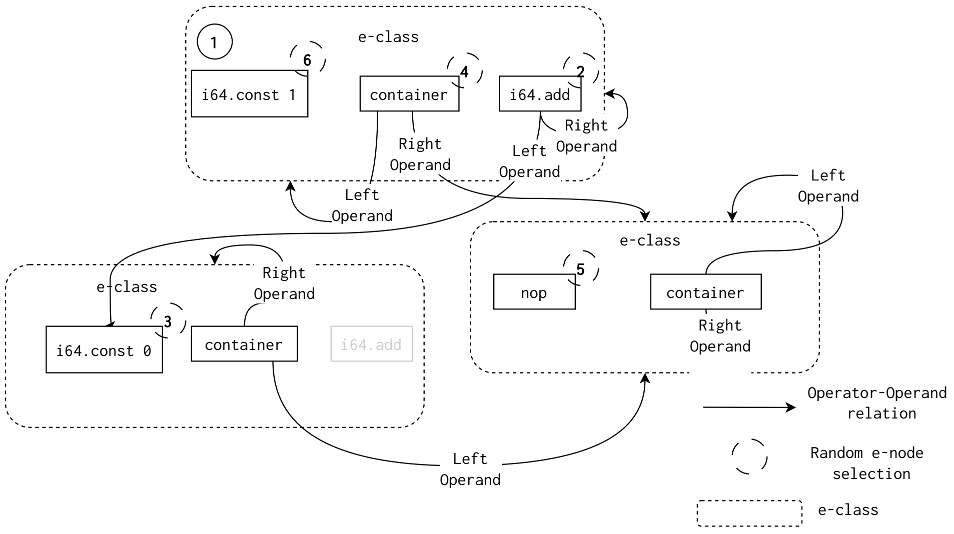

Figure 3.6: *e-graph built for rewriting the first instruction of Listing 3.8.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
       i64.const 1)
)
```

Listing 3.8: *Wasm function.*

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.add (
            i64.const 0
            i64.const 1
            nop
        ))
    )
```

Listing 3.9: *Random peephole mutation using egraph traversal for Listing 3.8 over e-graph Figure 3.6. The textual format is folded for better understanding.*

Leveraging the code presented in Listing 3.8 alongside the defined rewriting rules, we build the e-graph, simplified in Figure 3.6. In the figure, we highlight various stages of Algorithm 1 in the context of the scenario previously described. The algorithm initiates at the e-class with the instruction `i64.const 1`, as seen in Listing 3.8. At (2), it randomly selects an equivalent node within the e-class, in this instance taking the `i64.add` node, resulting: `expr`

`= i64.add l r`. As the traversal advances, it follows on the left operand of the previously chosen node, settling on the `i64.const 0` node within the same e-class ③. Then, the right operand of the `i64.add` node is choosen, selecting the `container` ④ operator yielding: `expr = i64.or (i64.const 0 container ( r nop ))`. The algorithm chooses the right operand of the `container` ⑤, which correlates to the initial instruction e-node highlighted in ⑥, culminating in the final expression: `expr = i64.or (i64.const 0 container( i64.const 1 nop)) i64.const 1`. As we proceed to the encoding phases, the `container` operator is ignored as a real Wasm instruction, finally resulting in the program in Listing 3.9.

Notice that, within the e-graph showcased in Figure 3.6, the container node maintains equivalence across all e-classes. Consequently, increasing the depth parameter in Algorithm 1 would potentially escalate the number of viable variants infinitely.

Contribution paper and artifact

WASM-MUTATE is fully presented in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" `https://arxiv.org/pdf/2309.07638.pdf`.

WASM-MUTATE is available at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate` as a contribution to the bytecodealliance organization [a].

[a] `https://bytecodealliance.org/`

3.4 Comparing CROW, MEWE, and WASM-MUTATE

In this section, we compare CROW, MEWE, and WASM-MUTATE, highlighting their key differences. These distinctions are summarized in Table 3.1. The table is organized into columns that represent attributes of each tool: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary for diversification, strength of the generated variants, and the security applications of these variants. Each row in the table corresponds to a specific tool. Notice that, the data and insights presented in the table are sourced from the respective papers of each tool and, from the previous discussion in this chapter.

Tool	Input	Core	Variants in 1h	Target	Variants Strength	Security applications
CROW	Source code or LLVM Ir	Enumerative synth.	> 1k	Code section	**96%**	Static analysis and side-channel attacks.
MEWE	Source code or LL VM Ir	CROW + Execution path randomization	> 1k	Code + Function sections	**96%**	Static and dynamic analysis, web timing-based attacks.
WASM-MUTATE	rewriting rules + **Wasm bin.**	e-graph random traversals	**$>$ 10k**	**Any Wasm part**	76%	Signature-based identification, static analysis, compiler fingerprinting and timing side-channel attacks.

Table 3.1: *Comparing CROW, MEWE and WASM-MUTATE. The table columns are: the tool's name, input format, core diversification strategy, number of variants generated within an hour, targeted sections of the WebAssembly binary, strength of the generated variants, and the security applications of these variants. The* Variant strength *accounts for the capability of each tool on generating variants that are preserved after the JIT compilation of V8 and wasmtime in average. Our three technical contributions are complementary tools that can be combined.*

CROW is a compiler-based strategy, needing access to the source code or its LLVM IR representation to work. Its core is an enumerative synthesis implementation with functionallity verification using SMT solvers, ensuring the functional equivalence of the generated variants. In addition, MEWE extends the capabilities of CROW, utilizing the same underlying technology to create program variants. It goes a step further by packaging the LLVM IR variants into a Wasm multivariant, providing MVE through execution path randomization. Both CROW and MEWE are fully automated, requiring no user intervention besides the input source code. WASM-MUTATE, on the other hand, is a semi-automated, binary-based tool. It needs a set of rewriting rules and the Wasm binary as inputs to generate program variants, centralizing its core around random e-graph traversals. Remarkably, WASM-MUTATE removes the need for compiler adjustments, offering compatibility with any existing WebAssembly binary.

We draw several interesting phenomena when aggregating the data presented in the corresponding papers of CROW, MEWE and WASM-MUTATE [**?** **?** **?**]. This can be appreciated in the fourth, fifth and sixth columns of Table 3.1. We have observed that WASM-MUTATE generates more unique variants in one hour than CROW and MEWE in at least one order of magnitude. This is mainly because WASM-MUTATE can generate variants in any part of the Wasm binary, while CROW and MEWE are limited to the code and function sections. In addition, CROW and MEWE generation capabilities are limited by the *overlapping* phenomenon discussed in Subsection 3.1.4. On the other hand, CROW and MEWE, by using enumerative synthesis, can exceed handcrafted optimizations [**?**], ensuring that the generated variants are preserved. In other words, the transformations generated out of CROW and MEWE are virtually irreversible by JIT compilers, such as V8 and wasmtime. This phenomenon is highlighted in the *Variants strength* column of Table 3.1, where we show that CROW and MEWE generate variants with 96% of preservation against 75% of WASM-MUTATE.

■ 3.4.2 Security applications

The last column of Table 3.1 highlights the security applications of the variants generated by our three technical contributions. Our tools generate many different and highly preserved code variants. This means that these variants, each with unique WebAssembly codes, maintain their distinctiveness even after JIT compilers translate them into machine codes (see Subsection 2.1.6). The preservation feature significantly reduces the impact of side-channel attacks that exploit specific machine code instructions, e.g., port contention [**?**]. Besides, the preserved transformations of the generated variants serve to reduce potential attack surfaces, thereby impeding signature-based identification.

Altering the layout of a WebAssembly program inherently influences its managed memory during runtime, a component not overseen by the WebAssembly program itself (see Definition 1). This pehomenon is especially important for

CROW and MEWE, given that they do not directly address the WebAssembly memory model. Significantly, CROW and MEWE considerably alter the managed memory by modifying the layout of the WebAssembly program. For example, the *constant inferring* transformations significantly alter the layout of program variants, affecting unmanaged memory elements such as the returning address of a function. Furthermore, WASM-MUTATE not only affects managed memory through changes in the WebAssembly program layout. It also adds rewriting rules to transform unmanaged memory instructions. Memory alterations, either to the unmanaged or managed memories, have substantial security implications, by eliminating potential jump points that facilitate malicious activities within the binary [**?**].

Besides, our technical contributions enhance security against timing-based attacks by creating variants that exhibit a wide range of execution times. This strategy is especially prominent in MEWE's approach, which develops multivariants functioning on randomizing execution paths, thereby thwarting attempts at timing-based inference attacks [**?**]. Adding another layer benefit, the integration of diverse variants into multivariants can potentially disrupt dynamic analysis tools such as symbolic executors [**?**]. Concretely, different control flows through a random discriminator, exponentially increase the number of possible execution paths, making multivariant binaries virtually unexplorable.

Key Takeaway

Our three technical contributions serve as complementary tools that can be combined to create a more comprehensive and robust software diversification strategy. For instance, when the source code for a WebAssembly binary is either non-existent or inaccessible, WASM-MUTATE offers a viable solution for generating code variants. On the other hand, CROW and MEWE excel in scenarios where high preservation is crucial, particularly when the generated variants may be subject to further analysis. Furthermore, WASM-MUTATE can benefit from the enumerative synthesis techniques employed by CROW and MEWE. Specifically, WASM-MUTATE could incorporate the transformations generated by these tools as rewriting rules.

■ 3.5 Conclusions

In this chapter, we discuss the technical specifics underlying our primary technical contributions. We elucidate the mechanisms through which CROW generates program variants. Subsequently, we discuss MEWE, offering a detailed examination of its role in forging MVE for WebAssembly. We also explore the details of WASM-MUTATE, highlighting its pioneering utilization of an e-graph traversal algorithm to spawn Wasm program variants. Remarkably, we undertake a comparative analysis of the three tools, highlighting their respective benefits and limitations, alongside the potential security applications of the generated Wasm variants.

In **??**, we present four use cases that support the exploitation of these tools. **??** serves to bridge theory with practice, showcasing the tangible impacts and benefits realized through the deployment of CROW, MEWE, and WASM-MUTATE.

04 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

■ 4.1 Offensive Diversification: Malware evasion

TODO The malware evasion paper

■ 4.1.2 Objective

Test and evade the resilience of WebAssembly malware detectors mentioned in Subsection 2.1.6.

■ 4.1.3 Approach

TODO We use wasm-mutate **TODO** How do we use it? **TODO** Controlled and uncontrolled diversification.

■ 4.1.4 Results

■ 4.2 Defensive Diversification: Speculative Side-channel protection

TODO Go around the last paper

■ 4.2.2 Threat model

- Spectre timing cache attacks.
    - Rockiki paper on portable side channel in browsers.

■ 4.2.3 Approach

- Use of wasm-mutate

[0] Comp. time 2023/10/03 14:18:41

- 4.2.4 Results

  - Diminshing of BER

05 CONCLUSIONS AND FUTURE WORK

■ 5.1 Summary of technical contributions

■ 5.2 Summary of empirical findings

■ 5.3 Summary of empirical findings

■ 5.4 Future Work

Moreover, the WebAssembly ecosystem is still in its infancy compared to more mature programming environments. A 2021 study by Hilbig et al. found only 8,000 unique WebAssembly binaries globally[**?**], a fraction of the 1.5 million and 1.7 million packages available in npm and PyPI, respectively. This limited dataset poses challenges for machine learning-based analysis tools, which require extensive data for effective training. The scarcity of WebAssembly programs also exacerbates the problem of software monoculture, increasing the risk of compromised WebAssembly programs being consumed[**?**]. This dissertation aims to mitigate these issues by introducing a comprehensive suite of tools designed to enhance WebAssembly security through Software Diversification and to improve testing rigor within the ecosystem.

[0] Comp. time 2023/10/03 14:18:41

REFERENCES

[1] A. Haas, A. Rossberg, D. L. Schuff, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, “Bringing the web up to speed with webassembly,” *PLDI*, 2017.

[2] P. Mendki, “Evaluating webassembly enabled serverless approach for edge computing,” in *2020 IEEE Cloud Summit*, pp. 161–166, 2020.

[3] M. Jacobsson and J. Wåhslén, “Virtual machine execution for wearables based on webassembly,” in *EAI International Conference on Body Area Networks*, pp. 381–389, Springer, Cham, 2018.

[4] “Webassembly system interface.” `https://github.com/WebAssembly/WASI`, 2021.

[5] D. Bryant, “Webassembly outside the browser: A new foundation for pervasive computing,” in *Proc. of ICWE 2020*, pp. 9–12, 2020.

[6] B. Spies and M. Mock, “An evaluation of webassembly in non-web environments,” in *2021 XLVII Latin American Computing Conference (CLEI)*, pp. 1–10, 2021.

[7] E. Wen and G. Weber, “Wasmachine: Bring iot up to speed with a webassembly os,” in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 1–4, IEEE, 2020.

[8] D. Lehmann, J. Kinder, and M. Pradel, “Everything old is new again: Binary security of webassembly,” in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020.

[9] M. Kim, H. Jang, and Y. Shin, “Avengers, assemble! survey of webassembly security solutions,” in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 543–553, 2022.

[10] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, “Crow: code diversification for webassembly,” in *MADWeb, NDSS 2021*, 2021.

[11] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, “Sledge: A serverless-first, light-weight wasm runtime for the edge,” in *Proceedings of the 21st International Middleware Conference*, p. 265–279, 2020.

[12] R. Gurdeep Singh and C. Scholliers, “Warduino: A dynamic webassembly virtual machine for programming microcontrollers,” in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, (New York, NY, USA), pp. 27–36, ACM, 2019.

[13] S. S. Salim, A. Nisbet, and M. Luján, “Trufflewasm: A webassembly interpreter on graalvm,” in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE ’20, (New York, NY, USA), p. 88–100, Association for Computing Machinery, 2020.

[14] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, “Wave: a verifiably secure webassembly sandboxing runtime,” in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2940–2955, 2023.

[15] Q. Stiévenart and C. De Roover, “Wassail: a webassembly static analysis library,” in *Fifth International Workshop on Programming Technology for the Future Web*, 2021.

[16] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, “Secwasm: Information flow control for webassembly,” in *Static Analysis* (G. Singh and C. Urban, eds.), (Cham), pp. 74–103, Springer Nature Switzerland, 2022.

[17] T. Brito, P. Lopes, N. Santos, and J. F. Santos, “Wasmati: An efficient static vulnerability scanner for webassembly,” *Computers & Security*, vol. 118, p. 102745, 2022.

[18] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, “Wasma: A static webassembly analysis framework for everyone,” in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 753–757, 2023.

[19] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, “Concolic execution for webassembly (artifact),” Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[20] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, “, : Sfi safety for native-compiled wasm,” *Network and Distributed Systems Security (NDSS) Symposium.*

[21] W. Fu, R. Lin, and D. Inge, “Taintassembly: Taint-based information flow control tracking for webassembly,” *arXiv preprint arXiv:1802.01050*, 2018.

[22] D. Lehmann and M. Pradel, “Wasabi: A framework for dynamically analyzing webassembly,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1045–1058, 2019.

[23] D. Lehmann, M. T. Torp, and M. Pradel, “Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly,” *arXiv preprint arXiv:2110.15433*, 2021.

[24] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, jan 2019.

[25] Q. Stiévenart, D. Binkley, and C. De Roover, "Dynamic slicing of webassembly binaries," in *39th IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2023.

[26] Q. Stiévenart, D. W. Binkley, and C. De Roover, "Static stack-preserving intra-procedural slicing of webassembly binaries," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 2031–2042, Association for Computing Machinery, 2022.

[27] K. Haßler and D. Maier, "Wafl: Binary-only webassembly fuzzing with fast snapshots," in *Reversing and Offensive-oriented Trends Symposium*, pp. 23–30, 2021.

[28] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1714–1730, 2018.

[29] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1129–1140, 2020.

[30] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system.," in *NDSS*, 2021.

[31] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*, pp. 122–142, Springer, 2018.

[32] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 313–326, 2018.

[33] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, pp. 840–852, 2019.

[34] D. Wang, B. Jiang, and W. Chan, "Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *arXiv preprint arXiv:2007.15510*, 2020.

[35] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "{EOSAFE}: security analysis of {EOSIO} smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1271–1288, 2021.

[36] Y. Huang, B. Jiang, and W. K. Chan, "Eosfuzzer: Fuzzing eosio smart contracts for vulnerability detection," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, pp. 99–109, 2020.

[37] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 703–715, 2022.

[38] S. Cao, N. He, Y. Guo, and H. Wang, "A general static binary rewriting framework for webassembly," *arXiv preprint arXiv:2305.01454*, 2023.

[39] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, (Los Alamitos, CA, USA), pp. 1101–1116, IEEE Computer Society, may 2022.

[40] N. Loose, F. Mächtle, C. Pott, V. Bezsmertnyi, and T. Eisenbarth, "Madvex: Instrumentation-based Adversarial Attacks on Machine Learning Malware Detection," *arXiv e-prints*, p. arXiv:2305.02559, May 2023.

[41] S. Cao, N. He, Y. Guo, and H. Wang, "WASMixer: Binary Obfuscation for WebAssembly," *arXiv e-prints*, p. arXiv:2308.03123, Aug. 2023.

[42] D. Chen and W3C group, "WebAssembly documentation: Security." `https://webassembly.org/docs/security/`, 2020. Accessed: 18 June 2020.

[43] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 119, 2018.

[44] Q. Stiévenart, C. De Roover, and M. Ghafari, "Security risks of porting c programs to webassembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, (New York, NY, USA), p. 1713–1722, Association for Computing Machinery, 2022.

[45] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Proc. of USENIX Security Symposium*, USENIX-SS'06, 2006.

[46] J. Cabrera Arteaga, "Artificial software diversification for webassembly," 2022. QC 20220909.

[47] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[48] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[49] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, "Multi-Variant Execution at the Edge," *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.

[50] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," in *Proceedings of the USENIX Security Symposium*, 2003.

[51] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[52] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.

[53] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly," *arXiv e-prints*, p. arXiv:2309.07638, Sept. 2023.

[54] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, jan 2021.

[55] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, IEEE, 2013.

[56] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova, "Babble: Learning better abstractions with e-graphs and anti-unification," *Proc. ACM Program. Lang.*, vol. 7, jan 2023.

[57] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, (New York, NY, USA), p. 264–276, Association for Computing Machinery, 2009.

[58] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, “Port contention goes portable: Port contention side channels in web browsers,” in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS ’22, (New York, NY, USA), p. 1182–1194, Association for Computing Machinery, 2022.

[59] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, “Swivel: Hardening WebAssembly against spectre,” in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1433–1450, USENIX Association, Aug. 2021.

[60] T. Schnitzler, K. Kohls, E. Bitsikas, and C. Pöpper, “Hope of delivery: Extracting user locations from mobile instant messengers,” in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, The Internet Society, 2023.

[61] A. Hilbig, D. Lehmann, and M. Pradel, “An empirical study of real-world webassembly binaries: Security, languages, use cases,” *Proceedings of the Web Conference 2021*, 2021.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus

Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

`https://doi.org/10.1145/3397537.3397567`

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus

Network and Distributed System Security Symposium (NDSS 2021), MADWeb

`https://doi.org/10.14722/madweb.2021.23004`

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry

Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)

https://dl.acm.org/doi/abs/10.1145/3560828.3564007

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023

`https://www.sciencedirect.com/science/article/pii/S0167404823002067`

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
Under revision

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry

11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

`https://doi.org/10.1145/3358504.3361228`